## CS61B Lecture #17



## Topics

standard Java Collections classes.

, ListIterators

s and maps in the abstract

nalysis of implementing lists with arrays.



## Data Types in the Abstract

time, should *not* worry about implementation of data
earch, etc.

o for us—their specification—is important.

eral standard types (in `java.util`) to represent collec-
cts

aces:

tion: General collections of items.

ndexed sequences with duplication

rtedSet: Collections without duplication

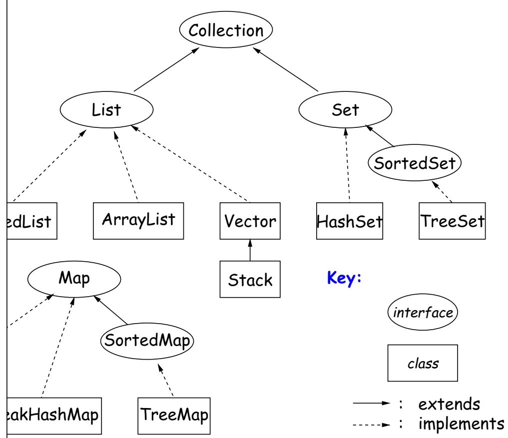
rtedMap: Dictionaries (key ↦ value)

classes that provide actual instances: `LinkedList`, `ArrayList`,
`TreeSet`.

hange easier, purists would use the concrete types only
nterfaces for parameter types, local variables.



## Collection Structures in java.util






## The Collection Interface

erface. Main functions promised:

hip tests: `contains` (∈), `containsAll` (⊆)

ries: `size`, `isEmpty`

`iterator`, `toArray`

modifiers: `add`, `addAll`, `clear`, `remove`, `removeAll` (set
e), `retainAll` (intersect)



## bout Library Design: Optional Operations

ections need to be modifiable; often makes sense just
from them.

rations are optional (`add`, `addAll`, `clear`, `remove`, `removeAll`,

developers decided to have *all* Collections implement
lowed implementations to throw an exception:

`UnsupportedOperationException`

e design would have created separate interfaces:

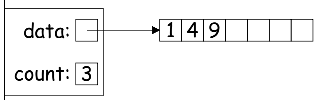
```
lection { contains, containsAll, size, iterator, ... }
andable extends Collection { add, addAll }
inkable extends Collection { remove, removeAll, ... }
ifiableCollection
llection, Expandable, Shrinkable { }
```

ave lots of interfaces. Perhaps that's why they didn't
y.

## nplementing Lists (I): ArrayLists

ncrete types in Java library for interface `List` are
nd `LinkedList`:

t expect, an `ArrayList`, `A`, uses an array to hold data.
, a list containing the three items 1, 4, and 9 might be
like this:




g four more items to `A`, its `data` array will be full, and
`data` will have to be replaced with a pointer to a new,
that starts with a copy of its previous values.

r best performance, how big should this new array be?

se the size by 1 each time it gets full (or by any con-
, the cost of $N$ additions will scale as $\Theta(N^2)$, which
`List` look much worse than `LinkedList` (which uses an
implementation.)



## Amortized Time

t the actual costs of a sequence of $N$ operations are
$_{-1}$, which may differ from each other by arbitrary amounts
$\in O(f(i))$.

ther sequence $a_0, a_1, \ldots, a_{N-1}$, where $a_i \in O(g(i))$.

$$\sum_{0\leq i<k} a_i \geq \sum_{0\leq i<k} c_i \text{ for all } k,$$

he operations all run in $O(g(i))$ *amortized time*.

actual cost of a given operation, $c_i$, may be arbitrarily
the amortized time, $a_i$, as long as the *total* amortized
ys greater than or equal to the total actual time, no
e the sequence of operations stops—i.e., no matter what

nterest, the amortized time bounds are much less than
dividual time bounds: $g(i) \ll f(i)$.

case of insertion with array doubling, $f(i) \in O(N)$ and



## ating Amortized Time: Potential Method

the argument, associate a *potential*, $\Phi_i \geq 0$, to the $i^{\text{th}}$
at keeps track of "saved up" time from cheap operations
"spend" on later expensive ones. Start with $\Phi_0 = 0$.

end that the cost of the $i^{\text{th}}$ operation is actually $a_i$, the
*ost*, defined

$$a_i = c_i + \Phi_{i+1} - \Phi_i,$$

ne real cost of the operation. Or, looking at potential:

$$\Phi_{i+1} = \Phi_i + (a_i - c_i)$$

erations, we artificially set $a_i > c_i$ so that we can in-
$_{+1} > \Phi_i$).

e ones, we typically have $a_i \ll c_i$ and greatly decrease $\Phi$
t it go negative—may not be "overdrawn").

all this so that $a_i$ remains as we desired (e.g., $O(1)$ for
ray), without allowing $\Phi_i < 0$.

t we choose $a_i$ so that $\Phi_i$ always stays ahead of $c_i$.



## The List Interface

`lection`

represent *indexed sequences* (generalized arrays)

thods to those of `Collection`:

ip tests: `indexOf`, `lastIndexOf`.

`get(`$i$`)`, `listIterator()`, `sublist(`$B, E$`)`.

: `add` and `addAll` with additional index to say *where* to
wise for removal operations. `set` operation to go with

`erator<Item>` extends `Iterator<Item>`:

`vious` and `hasPrevious`.

`ve`, and `set` allow one to iterate through a list, inserting,
or changing as you go.

**t Question:** What advantage is there to saying `List L`
an `LinkedList L` or `ArrayList L`?



## Expanding Vectors Efficiently

array for expanding sequence, best to *double* the size
row it. Here's why.

ze $s$, doubling its size and moving $s$ elements to the new
time proportional to $2s$.

there is an additional $\Theta(1)$ cost for each addition to
actually assigning the new value into the array.

ld up these costs for inserting a sequence of $N$ items,
t turns out to be proportional to $N$, as if each addition
t time, even though some of the additions actually take
ional to $N$ all by themselves!



## ortization: Expanding Vectors (II)

| | Resizing Cost | Cumulative Cost | Resizing Cost per Item | Array Size After Insertions |
|---|---|---|---|---|
| | 0 | 0 | 0 | 1 |
| | 2 | 2 | 1 | 2 |
| | 4 | 6 | 2 | 4 |
| | 0 | 6 | 1.5 | 4 |
| | 8 | 14 | 2.8 | 8 |
| | 0 | 14 | 2.33 | 8 |
| | ⋮ | ⋮ | ⋮ | ⋮ |
| | 0 | 14 | 1.75 | 8 |
| | 16 | 30 | 3.33 | 16 |
| | ⋮ | ⋮ | ⋮ | ⋮ |
| | 0 | 30 | 1.88 | 16 |
| | ⋮ | ⋮ | ⋮ | ⋮ |
| $-1$ | 0 | $2^{m+2}-2$ | $\approx 2$ | $2^{m+1}$ |
| | $2^{m+2}$ | $2^{m+3}-2$ | $\approx 4$ | $2^{m+2}$ |

out (*amortize*) the cost of resizing, we average at most
e units for resizing on each item: "amortized resizing
ts." Time to add $N$ elements is $\Theta(N)$, *not* $\Theta(N^2)$.

## Application to Expanding Arrays

- to our array, the cost, $c_i$, of adding element #$i$ when
eady has space for it is 1 unit.
- es not initially have space when adding items 1, 2, 4, 8,
ner words at item $2^n$ for all $n \geq 0$. So,
- $\geq 0$ and is not a power of 2; and
- $1$ when $i$ is a power of 2 (copy $i$ items, clear another $i$
d then add item #$i$).
- operation #$2^n$ we're going to need to have saved up at
$^{n+1}$ units of potential to cover the expense of expanding
nd we have this operation and the preceding $2^{n-1} - 1$
which to save up this much potential (everything since
g doubling operation).
- = 1 and $a_i = 5$ for $i > 0$. Apply $\Phi_{i+1} = \Phi_i + (a_i - c_i)$, and
happens:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 9 | 1 | 1 | 1 | 17 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 33 | 1 |
| 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 6 | 2 | 6 | 10 | 14 | 2 | 6 | 10 | 14 | 18 | 22 | 26 | 30 | 2 |

*Pretending each cost is 5 never underestimates true cumulative time.*